Neural Programming and an Internal Reinforcement Policy *

Astro Teller and Manuela Veloso

Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract. An important reason for the continued popularity of Artificial Neural Networks (ANNs) in the machine learning community is that the gradient-descent backpropagation procedure gives ANNs a locally optimal change procedure and, in addition, a framework for understanding the ANN learning performance. Genetic programming (GP) is also a successful evolutionary learning technique that provides powerful parameterized primitive constructs. Unlike ANNs, though, GP does not have such a principled procedure for changing parts of the learned system based on its current performance. This paper introduces *Neural Programming*, a connectionist representation for evolving programs that maintains the benefits of GP. The connectionist model of Neural Programming allows for a regression credit-blame procedure in an evolutionary learning system. We describe a general method for an informed feedback mechanism for Neural Programming, *Internal Reinforcement*. We introduce an Internal Reinforcement procedure and demonstrate its use through an illustrative experiment.

1 Introduction

This paper introduces a new representation for learning complex programs. This new representation, *Neural Programming* (NP) has been developed with the goal of incorporating positive aspects of both artificial neural networks and genetic programming. Neural Programming is a connectionist programming language that has been designed to make *internal reinforcement*, hither-to unaccomplished in genetic programming, possible.

Both GP [6] and ANNs [9] continue to be well investigated fields. In ANNs, the focus on improving the power of the technique has not been on changing what is inside an "artificial neuron." Works like [4, 10] have, however, investigated the possible additional benefit of complicating and un-homogenizing artificial neurons. To the best of our knowledge, in the context of ANNs and principled update policies like backpropagation, these investigations have not yet extended to arbitrary, potentially non-differentiable functions like those typically used by human programmers and by evolving GP programs.

In GP, the focus of investigation for increased power of the technique has not been on changing the GP representation or on finding principled (non-random) update policies. Some work, has, however, been done in these areas. [8] describes a process for trying to find sub-functions in an evolving GP function that are more likely than randomly selected ones to contribute positively to fitness when crossed-over into other programs. [3] and [5] describe possible approaches for allowing the mechanism of evolution to

* This work is supported in part through an ONR Muri grant and the first author is supported through the generosity of the Hertz Foundation.

provide self-adaptation all the way down to the single node level. For example, in a finite state machine or a GP program, each node can express an evolved preference for interaction with the update strategies like mutation and crossover.

In brief, ANN is a successful representative of the machine learning practice of *explicit credit-assignment*. GP is a successful representative of *empirical credit-assignment* [2]. Empirical credit-assignment is the machine learning practice of allowing the dynamics of the system to implicitly determine credit and blame. Evolution does just this [1]. The goal of this work is to begin to bridge this credit-assignment gap by finding ways in which explicit and empirical credit-assignment can find mutual benefit in a single machine learning technique.

In general, it is not possible to give an ANN an input for every possible parameterization of each user defined primitive function a GP program can be given. And it is far from obvious how to work complex functions into the middle of an otherwise homogeneous network of simple non-linear functions. Yet, gradient-descent learning procedures, like backpropagation in ANNs, are an extremely powerful idea. Backpropagation is not only a kind of performance guarantee, it is a kind of performance explanation. It is to achieve this kind of dual benefit that the research this paper reports was undertaken.

We show in this paper how we can accumulate explicit credit-assignment information in Neural Programming. These values will be collectively referred to as the *Credit-Blame Map*. A representation that would facilitate a credit-blame map would accomplish two things. First, by looking over a program's credit-blame map, humans could, with considerably less work, sort out *what* a particular successful program was doing and thereby increase the interpretability of GP programs. Second, by organizing the GP programs in a network of nodes but by replacing program flow of *control* with flow of *data*, we can use the credit-blame map to propagate punishment and reward through the program, getting the benefits from both the GP and ANN worlds. In short, it would be desirable to be able, in GP, to have reinforcement of the programs be more specific (directed towards particular parts or aspects of a program) and more appropriate (telling the system *how* to change those specific parts). ANNs, on the other hand, might benefit greatly from the use of the kind of program primitives that are natural to GP and often found to be advantageous to the learning process.

This paper will do three things. First, we will contribute the Neural Programming representation as a new, connectionist, representation in which to evolve programs. Second, this paper will describe how this representation can be used to deliver explicit, useful, internal reinforcement to the evolving programs. And third, an experiment on a illustrative signal classification problem will be used to demonstrate the promise of both the representation and its associated internal reinforcement strategy.

2 Neural Programming

The essence of a programming language is one or more basic constructs and one or more legal ways of combining those constructs. A measure of the extendibility of a language is the ease with which new constructs or new construct combinations can be incorporated into the language. It is the high degree of extendibility in GP that we want to wed to the focused update policies possible in ANNs.

The Neural Programming representation has flow of data, like an ANN, rather than the flow of control typical in programs. The nodes in a neural program can be arbitrary functions of the inputs. So a node can still be an average of the inputs to the node and a sigmoid threshold. But it can also other functions such as ADD, READ, WRITE, IF-THEN-ELSE, and, most importantly, potentially complex user defined functions for examining the input data (see figure 1). Such input-access-primitives might, for example, return the AVERAGE or VARIANCE of values in a range of the input data as specified by the inputs to that node. This kind of embedding of complex (often co-evolved) components as primitives in the evolving GP system has repeated shown to be effective (e.g., [7]) and is an important reason for trying to find a compromise between GP and ANNs. Furthermore, these powerful input-access-primitives, as part of the learning process, can be used in place of brittle preprocessing.

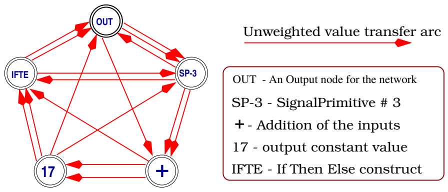

Fig. 1.: A simple program in the NP representation

The multiple forked distribution of good values from any point in the program is probably a valuable aspect of the NP representation. In the parley of GP, this can be seen as a kind of highly flexible automatically defined function (ADF) [7] mechanism. This fan-out is a connectionist advantage that GP programs might profit by incorporating.

Neural Programming was designed to be a representation for *evolving* programs. As such, there are two dominate forms of change that evolving programs typically undergo: crossover and mutation. Mutation is the change of one (usually atomic) part of the program to another aspect of the same type. Crossover is the sexual reproduction of two programs; two programs "mate" by exchanging program material between them.

> *While NP programs look more like recurrent ANNs than GP programs, NP programs are changed,* **not** *by adjusting arc weights (NP arcs have no weights), but by changing what is* **inside** *each node, the program* **topology** *and* **size***.*

Here are the major characteristics of the Neural Programming representation:

- A Neural Program consists of nodes and arcs.
- Each node K has K_i input and K_o output arcs.
- Each node may be one of a set of functions (e.g., Read, Write[2], Multiply, Input-Access-Primitive-3, etc.) or zero-arity functions (e.g. constants, *clock*, etc.).
- On **each** time step t ($0 < t < MaxTimeSteps$), **every** node takes some or all of its inputs, computes its node function of those inputs, and then outputs the value to *all* of its output arcs. *This is a network flow style program!*

[2] These memory functions manipulate an indexed, randomly addressable memory [11].

- One of the possible node functions is "OUTPUT." OUTPUT nodes are responsible for contributing, through a function F_1 of the inputs and the time, to the program's response collected in OutputTotal.
- There can be multiple "OUTPUT" nodes in one program (as determined by evolution). They all contribute to the same program variable, OutputTotal.
- When the program finishes ($t = MaxTimeSteps$) the program's response is set to a function of the OutputTotal variable.[3]

3 Internal Reinforcement in Neural Programming

Now that we have this neural programming representation, we introduce a mechanism to accomplish internal reinforcement. In the Internal Reinforcement of Neural Programs (IRNP), there are two main stages. The first stage is to classify each node and arc of a program with its perceived contribution to the program's output. This set of labels will be collectively referred to as the *Credit-Blame map*. The second stage is to use this Credit-Blame map to change the program in a way that is more likely than random to improve its performance.

It is still an open question in our research which methods to use to best accomplish the goals of internal reinforcement. We have already identified several methods for accomplishing each of the two stages. For the sake of brevity and clarity, this paper will focus on only one technique for each of the two stages.

The evolved NP programs that we consider now are part of the PADO system (described in [12, 13, 14]). For the purpose of this paper, it is sufficient to know that PADO is a machine learning system designed for signal classification. In PADO, an evolving program need only learn to discriminate one signal from the others in order to survive and be useful to the PADO system. The experiments we report to illustrate NP and IRNP are run in PADO.

3.1 Creating a Credit-Blame map

Most machine learning problems can either be described as target value prediction or classification problems. There are a number of ways in which classification problems can be reduced to target value prediction problems. In PADO this is done by evolving "discriminator" programs for each of the classes and then orchestrating their responses. This means that for each program in PADO, the value it should return is ordinal (i.e. if v is the right value to return, then a program that says $v - 1$ is better than a program that says $v - 2$). Having collapsed these two parts of machine learning into one view, let us consider an abstract input to output mapping to be learned by the neural programs.

Accumulate Explicit Credit-Assignment Values For each node X, over all time steps on a particular training example i, we will accumulate some function[4] of the values node X outputs into a value V_X^i. We will call the correct answer (the correct target value) for training instance i, T^i.

[3] NP programs can also return answers through memory, rather than through OUTPUT nodes.

[4] For this paper, this will be the mean of the values but other functions are being investigated.

Now we have two continuous variables in i: T^i and V_X^i. We can compute the correlation between them. We will call this the *credit score* for node X, notated as CS_X.

With CS_X, we already have a simple Credit-Blame map: a value associated with each node in the program that helps indicate its contribution to the program. A neural program such that every node X has a value of zero for CS_X is, by definition, a useless program. The reader should, however, be concerned with a situation in which nodes X and Y produce values and node Z computes an XOR of these two values. In this case, even if Z has a high credit score, X and Y will not. And yet it is clear that nodes X and Y are largely responsible for node Z's success. We can continue to refine the Credit-Blame map to attend to this type of indebtedness relationships.

Refine Credit-Assignment Values by Bucket-Brigade We can now use the topology of the NP program, in a bucket-brigade style backward propagation, to refine the credit scores at each node. There is a wealth of literature on propagation of values through networks. Our current research includes an investigation into an understanding the effects of these various methods so that we can incorporate the most effective paradigm into the IRNP mechanism.

For the experiment described in this paper we will use the simplest mechanism possible: each node X will be given the *maximum* of the credit scores of nodes such that one of the outputs of X is an input to that node and that node was effected by the values output by X. If this were a highly connected graph and the arity of all nodes was equal to the number of their inputs, then this bucket-brigade operation would simply fill the entire Credit-Blame map with the maximum value in the map. In practice, this does not happen exactly because the neural programs have many arcs whose values are ignored (even though very few of the node values are ignored). For example, a node Y with a function value of the constant value "6" outputs its value at each time step and ignores all values it receives as input. So even if this constant is relevant to a node with a high credit score (and so node Y, in turn, receives a high credit score) the nodes whose outputs are inputs to Y will receive no boost in credit score from this source.

In general, the arcs in an NP program are given credit scores that differ from either their source or destination nodes (though this new value is a function of the two node credit scores). For this paper, we will simply assign each arc from node X to node Y the credit-score pair (CS_X,CS_Y).

3.2 Using a Credit-Blame map

The second phase of the internal reinforcement is the use of the created Credit-Blame map to enhance the probability that the program updates will lead either to a better solution or to a similar solution in less time. There are two basic ways that the Credit-Blame map can be used to do this enhancement: through improvement of either the mutation or crossover strategies.

The possibility of using internal reinforcement (explicit credit assignment) not only for mutation (which has analogies to the world of ANNs) but for crossover as well is important. Traditional GP uses random crossover and relies entirely on the mechanism of empirical credit assignment. Work has been done to boot-strap this mechanism by using the evolutionary process itself to evolve improved crossover procedures (e.g. [3, 12]).

Thus, NP has the potential not only to improve on the existing GP mechanism, but also to help *explain* the central mystery of GP, crossover.

Mutation can take a variety of forms in NP. These various mutations can be best divided as: add an arc, delete an arc, swap two arcs, change a node function, add a node to the program, delete a node from the program. In the simple experiment shown in the next section each of these mutations took place with equal likelihood in both the random and internal reinforcement recombination cases. For example, to add an arc under random mutation to an NP program we simply pick a source and destination node at random from the program to be mutated and add the arc between the nodes.

Internal reinforcement can have a positive effect on this recombination procedure. For each recombination type, we pick a few nodes (eight in this paper's experiment) at random and pick a node or arc (depending on the mutation type) that has maximal or minimal credit score as appropriate. For example, when deleting a program node, we can pick a few nodes at random and delete the node with the lowest credit score.

In the random version of crossover, we simply pick a "cut" from each graph (i.e., a subset of the program nodes) at random and exchange and reconnect them. Like the mutation, we will keep the same underlying mechanism (for this paper) and simply try to get "good" program fragments to change. Given that we will separate a program into two fragments before crossover, let us define *CutCost* to be the sum of all credit scores of *inter*-fragment arcs and *InternalCost* to be the sum of all credit scores of *intra*-fragment arcs in the program to be crossed-over. For these definitions we will take the credit score of an arc to be the credit score of its destination node. Now we will say that the cost of a particular fragmentation of a program is equal to *CutCost*/(*InternalCost* + *CutCost*). If we try to minimize this value for both of the program fragments we choose, we are much less likely to disrupt a crucial part of the program during crossover.

In addition to these "aids to random search", we introduce specific changes to a program that can be affected with the use of the Credit-Blame map. For example, the IRNP procedure may promote high sensitivity parts of programs, but may not necessarily connect these parts to the output nodes. We can refine a program in the following way. First, let X be the node with highest CS_X. Second, let u be the output arc of X to a node Y that minimizes CS_Y. Third, for all arcs p that have an OUTPUT node as a destination, pick the arc v whose source node X has minimal CS_X. Finally, switch arcs u and v.

4 Experimental Results

This section describes an experiment using the IRNP paradigm. In order to focus on the issues at hand, a set of *simple* manufactured inputs is used. Figure 2 shows an example input from each of the three classes in this classification problem.

The node functions used were $\{$constants (0..256), *clock*, +, -, *, READ, WRITE, POINT, AVERAGE$\}$. POINT is an input access primitive that uses one input arc as a parameter p_1 and returns Input[p_1]. AVERAGE is an input access primitive that uses two input arcs as parameters (p_1,p_2) and returns the mean of values in $\{$Input[p_1]..Input[p_2]$\}$.

Figure 3 shows the classification performance of the best evolved programs. This experiment was run on a small population (210 individuals). Each individual was "pulsed" 20 times before a response was retrieved through the OUTPUT nodes. The training took

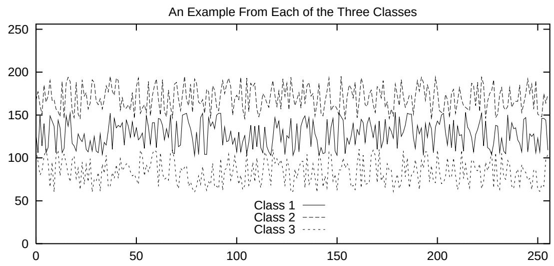

Fig. 2.: Signal examples: All training and testing examples are random vectors of length 256 with class centerlines at 171, 128, and 85 and a point independent standard deviation of 12.

place with 210 randomly generated instances and the separate testing set was made up of 900 randomly generated instances. This particular experiment used a mutation percentage rate of 70% and a crossover percentage rate of 25%. It is worth noting that internal reinforcement only adds about 6% to the computational expense of a generation.

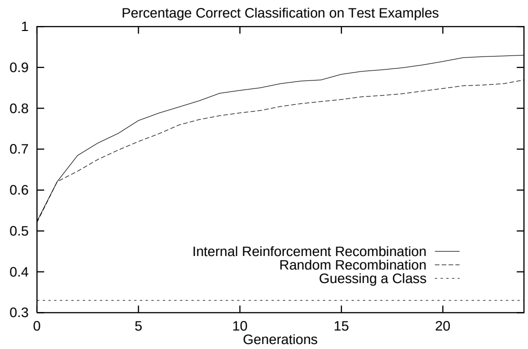

Fig. 3.: This graph compares the classification performance of Neural Programming with and without Internal Reinforcement on the signal classes outlined above on an out-of-training test set.

The point of this section is to demonstrate that *internal reinforcement* can have a significant positive effect on the evolution of algorithms. The experiment shown in figure 3 displays this effect; the curve plots the generalization classification ability of the best NP programs on that generation, averaged over 100 runs on a test set of previous unseen, randomly generated inputs. To focus on the immediate benefit of internal reinforcement, the graph above shows only the first 25 generations of computation. This is a simple problem and both curves reach 100% before generation 50. Over these first 25 generations, the Neural Programming population using IRNP learns better than the NP population with standard recombination. On average, at each generation, the IRNP programs generalize to unseen examples better than the NP population using random mutation at the same generation. More significantly, for all levels of generalization performance that both populations tend to reach, the NP populations evolved under IRNP reach that level, on average, in about 30% of the generations.

5 Conclusions

This paper has presented a new representation for learning complex programs. This new representation, *Neural Programming* has been developed with the goal of incorporating positive aspects of both artificial neural networks and genetic programming. Neural Programming is a connectionist programming language which has been designed to make *internal reinforcement*, hither-to unaccomplished in genetic programming, possible. A simple way of accomplishing this sort of internal reinforcement was detailed and some alternatives and extensions were briefly mentioned. We illustrated the technique with a focused experiment that showed that internal reinforcement improves the speed and accuracy of Neural Programming learning.

Neural Programming and associated internal reinforcement policies are our on-going research. The goal of this paper has been to communicate the exciting possibility that, through the exploration of new program representations, we may be able to find ways to capture the explanation and update power of backpropagation with the flexibility and generality of genetic programming.

References

1. L. Altenberg. The evolution of evolvability in genetic programming. In Jr. Kenneth E. Kinnear, editor, *Advances In Genetic Programming*, pages 47–74. MIT Press, 1994.
2. P. Angeline. *Evolutionary Algorithms and Emergent Intelligence*. PhD thesis, Ohio State University, Department of Computer Science, 1993.
3. P. Angeline. Two self-adaptive crossover operators for genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*. MIT Press, 1996.
4. F. Dellaert and R.D. Beer. Co-evolving body and brain in autonomous agents using a developmental model. In *Technical Report CES-94-16, Department of Computer Engineering and Science*. Case Western Reserve University, Cleveland, OH 44106, 1994.
5. L. Fogel, P. Angeline, and D. Fogel. An evolutionary programming approach to self-adaptation on finite state machines. In J. McDonnell, R. Reynolds, and D. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*. MIT Press, 1995.
6. J. Koza. *Genetic Programming*. MIT Press, 1992.
7. J. Koza. *Genetic Programming 2*. MIT Press, 1994.
8. J. Rosca and D. Ballard. Discovery of subroutines in genetic programming. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*. MIT Press, 1996.
9. D.E. Rumelhart, G.E. Hinton, and R.J Williams. Learning internal represenations by error propogation. In *Parallel Distributed Processing*. MIT Press, Cambridge, MA, USA, 1986.
10. K. Sharman, A. Alcazar, and Y. Li. Evolving signal processing algorithms by genetic programming. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, 1995.
11. A. Teller. The evolution of mental models. In Kenneth E. Kinnear, editor, *Advances In Genetic Programming*, pages 199–220. MIT Press, 1994.
12. A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In K. Kinnear and P. Angeline, editors, *Advances in Genetic Programming 2*. MIT, 1996.
13. A. Teller and M. Veloso. Program evolution for data mining. In Sushil Louis, editor, *The International Journal of Expert Systems. Third Quarter. Special Issue on Genetic Algorithms and Knowledge Bases.*, pages 216–236. JAI Press, 1995.
14. A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*. Oxford University Press, 1996.

This article was processed using the LaTeX macro package with LLNCS style